Distributed Systems Notes

Section 1: Introduction

What is a Distributed System:

- A collection of independent computers that appears to its users as a single coherent system.
    - Number of components
    - Communication between components (message passing)
    - Achieve more than each machine can individually

A computer network is just a collection of computers passing messages. E.g the internet is a computer network. It does not appear as a single system to users.

Advantages of a distributed System:

- Resource sharing (disks, printers, files, processing power)
- Availability, scalability, reliability.

Disadvantages of a distributed System:

- Concurrency
- Heterogeneity
- No global clock
- Independent failures

The internet:

The internet is a large number of interconnected computer networks. Includes features such as protocol for message passing and has services like world wide web(www) and email and file transfer.

Intranets:

Portion of internet that is separately administered by organizations. Connected to internet via a router. Firewalls to protect intranet from unauthorized messages.

Distributed System Challenges:

Heterogeneity: Distributed systems use hardware and software resources of varying characteristics. E.g OS is linux or windows, or programming language. Can solve problem by using middleware or agreeing on standard data/protocols.

Middleware: software layer between the distributed application and operating systems. E.g Distributed file system, RPC (procedural language), RMI. (Object-oriented language).

Openness: Ability to extend the system in different ways be adding hardware or software resources. E.g adding interfaces.

Security:

- Confidentiality: Protect against unauthorized users
- Integrity: Protect against alteration and corruption
- Availability: protection against interference with access

Security mechanisms include encryption, authentication (passwords, keys), authorization (access control)

Scalability: System needs to handle the growth of users and data.

- Cost of physical resources
- Controlling performance loss, avoiding bottle necks (good algorithms)
- Available resources

Failure Handling:

- Detecting, Masking (message retransmission), Tolerating (report failure to user), recovery (fix corrupted data).

Concurrency: Multiple clients accessing the same resource at same time. (Use semaphores)

Transparency: Hiding the components of a distributed system for the user and application programmer. (Access, location, concurrency, failure and scaling transparency)

Section 2: Models

Communication Paradigms (Low level to high level):

Interprocess communication: Multicast (message to multiple users), socket con.

Remote Invocation: Call a remote operation between dist. entities. RMI, RPC.

Indirect communication: Space uncoupling (senders don't know who's sending). Time uncoupling (senders and receivers don't need to exist at same time).

Roles and Responsibilities:

Client: Initiates connection to other process.

Server: Receives connection, offers service.

Peer: Client or server, connect to and receive connection from other peers.

Placement:

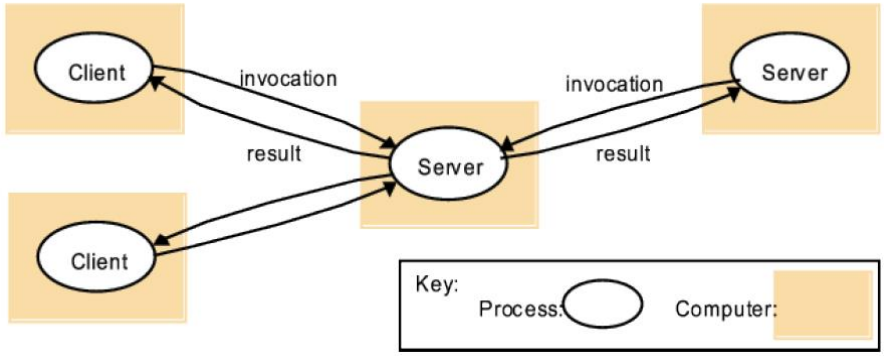
Caching: Storing data at places closer to client speeds up responses.

Mobile code: Transferring code to location most efficient. E.g running complex query on another machine. Forcing client to execute code etc.

Mobile agents: Code and data together executes on the client PC. Also, agent may check for updates to ensure software on client's PC is up to date.

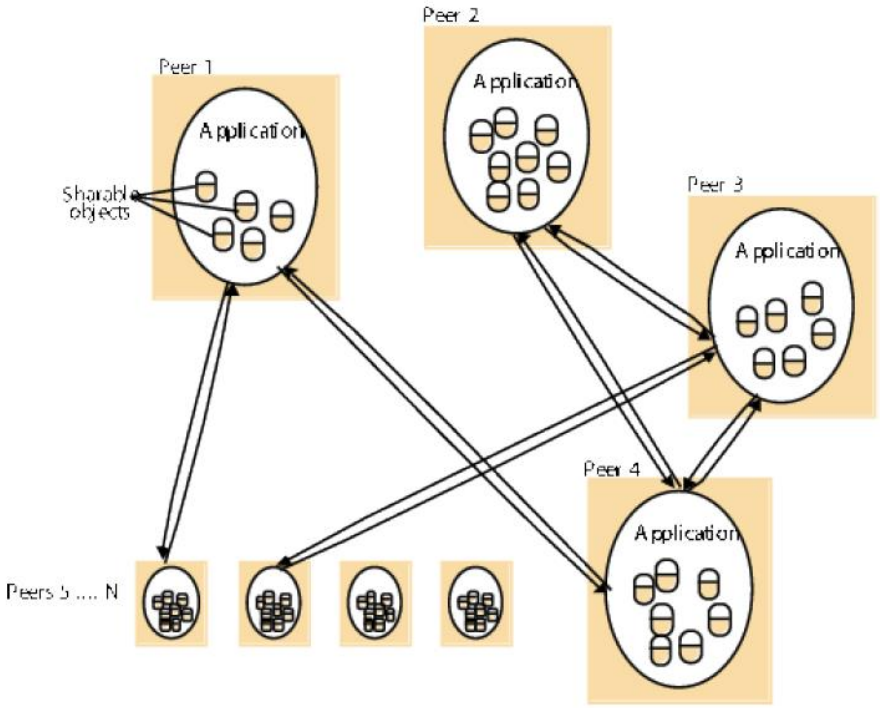
Architectural Patterns:

Client-Server: Client invokes services in server and results are returned.



Server and a bunch of clients who can make use of the server. ***Client's don't talk to one another****. Communication happens through the server.*

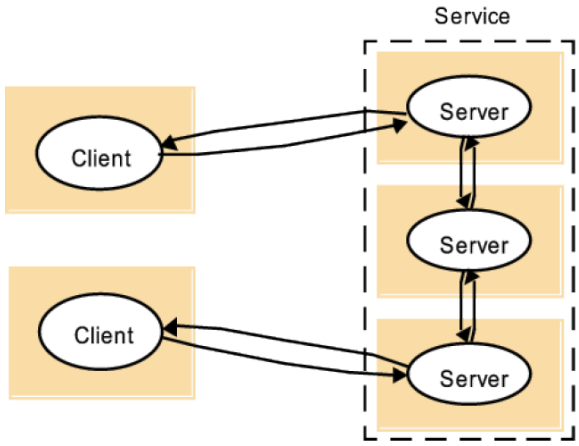
Peer-to-Peer: Each process in the system plays a similar role as client or server.



Every component can make a connection to every other component as well as receive a connection. ***There is no central server****. Can be* ***more secure than client-server*** *as it is* ***not susceptible to a single point of failure****. Bringing down all peers harder than a central server. Peer is not more important than another peer.*

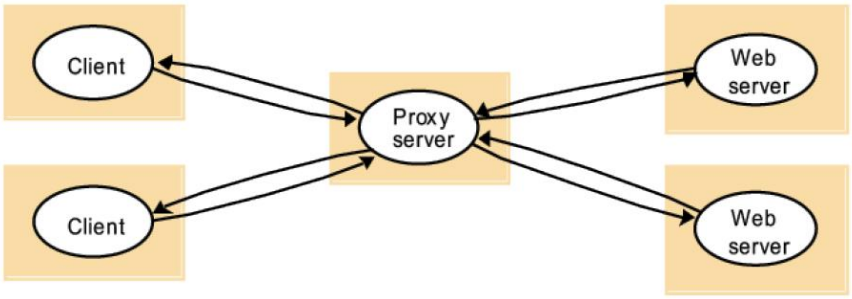
Distributed Architecture Variations:

Service Provided by multiple servers:



- *Objects may be replicated across servers*
- *Helps with load balancing*
- *Can offer service closer to client*
- *Survives failure of one server*

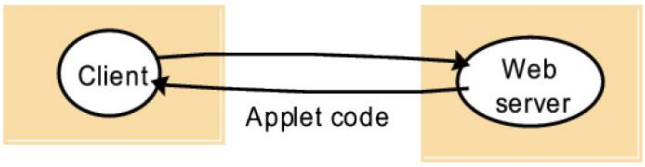
Proxy servers and caches:



- *Cache is a store of recently used objects closer to client*
- *Helps with load balance*
- *Speeds requests up*
- *Can use in authentication*

Mobile Code and Agents:

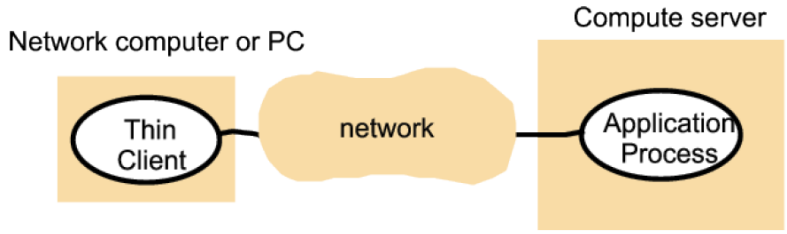
a) client request results in the downloading of applet code



b) client interacts with the applet

- *Downloaded and executed by the client.*
- *Less resources used by server -> scalable.*

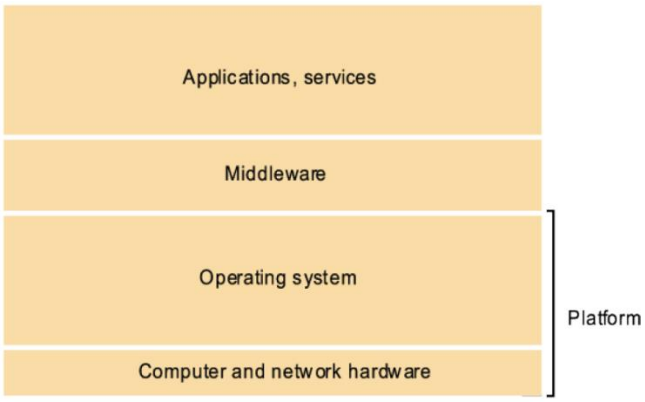
Network Computers and Thin Clients:



Network Computers: Download their OS and application software from a remote file system. Applications run locally.

Thin Clients: Application is not downloaded but runs on the computer server. On our end it jus displays information and takes input. Run apps in the cloud.

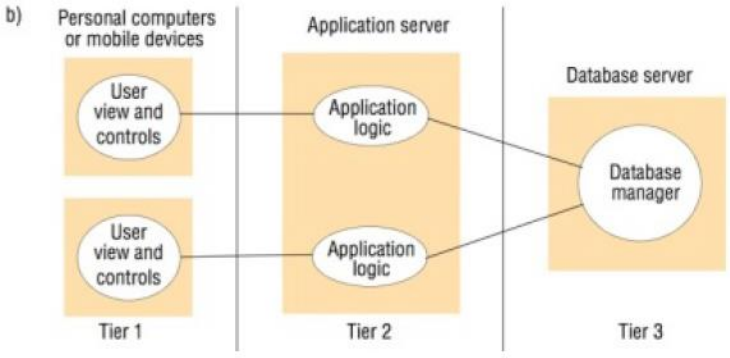
Layers:



Middleware:

- Value adding services such as naming, security, event service.
- Adds overhead due to additional level of abstraction.

Tiered Architecture:



Fundamental Models:

Non-functional aspects of the distributed system. E.g Reliability, security, performance.

<u>Interaction Model:</u>

Transmission of messages, performance of communication channels, computer clocks and timing events.

Latency (performance): Delay in data transfer.

Bandwidth (performance): Amount of data that can be transmitted in fixed time.

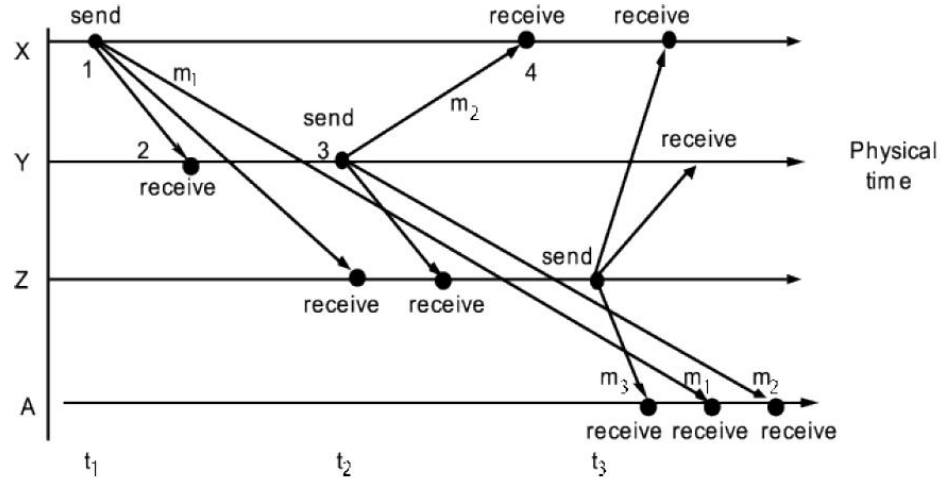
Jitter (performance): Latency variation in packet flow.

Event Timing: Components can have different timestamps. Difference in time set.

Synchronous System Model (Interaction Model): Bounds on the time to execute each step of a process, bound on message transmission day, bound on clock drift.

Asynchronous System Model (Interaction Model): If a communication finishes quickly you can move on to the next round if available. No bounds on Process execution speed, message transmission delays, clock drift rates.

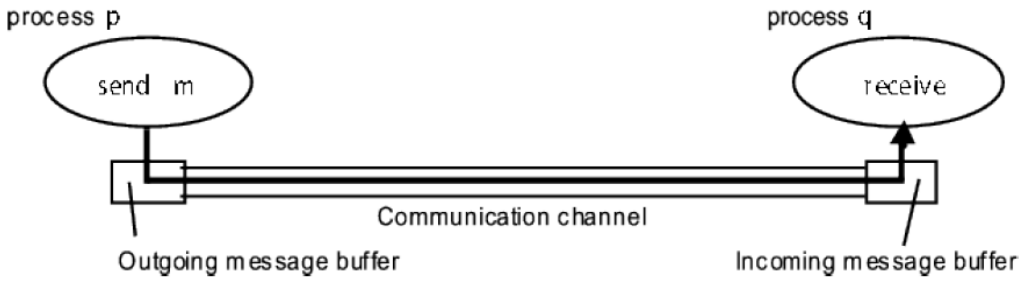
Event Ordering:



Events can be interpreted based on sequence of event numbers as opposed to time of receipt. Picture shows how long communications took. Can't receive something from future so draw arrows in right direction. Exam? Maybe? This diagram can help with debugging sequential behavior.

Failure Model:

Omission Failures: Process or communication channel fails to perform what is expected. Caused by process crash, timeouts etc. Example: read, write, buffer



Send Omission failure: Message not sent from sender to out-going buffer.

Receive Omission failure: Message not received in receiver incoming buffer.

Channel Omission failure: Message not being transported through channel.

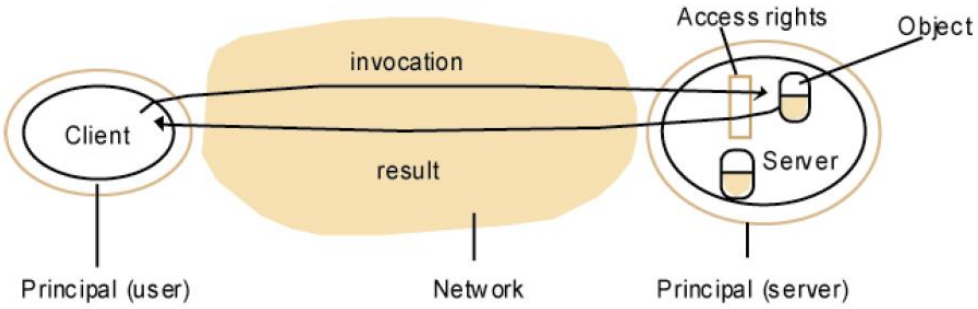
Arbitrary Failures: Message could be corrupted. Messages delivered more than once. Steps in process missed.

Class of failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a send, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing Failures: Occur when limits set on process execution time, delivery time and clock rate drift are not met. More relevant to synchronous systems than asynchronous.

Security Model:

Protecting Objects:



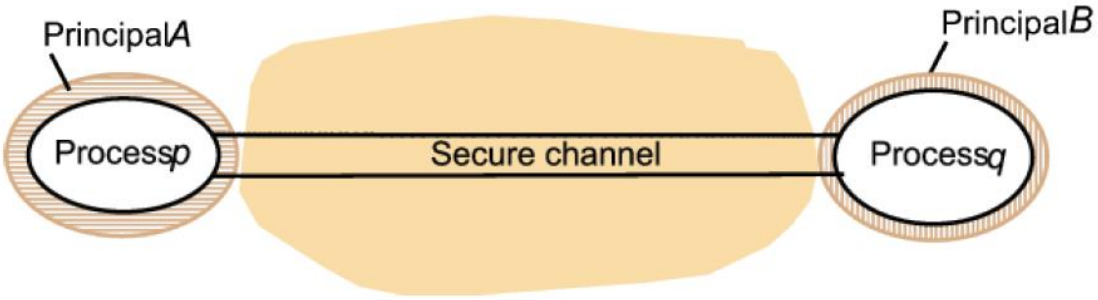
Security is achieved by securing processes, communication channels and preventing unauthorized access.

Access rights: who can perform operations on an object.

Enemy Threats:

- Spoofing source address of processes.
- Threat to communication channel.
- Denial of service attack.

Addressing the Threat: Encryption, authentication.



Reliability of one-to-one communication defined by:

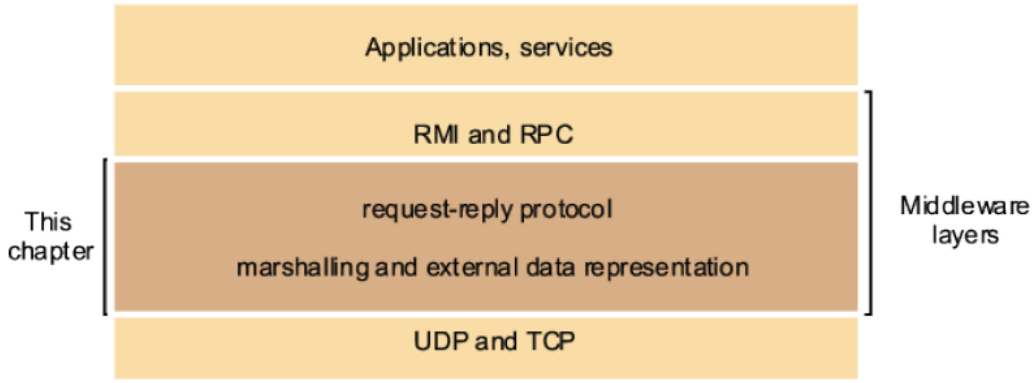
Validity: Message in outgoing buffer is delivered to incoming buffer.

Integrity: Message is identical to the one sent, and no messages delivered twice.

Section 3: Interprocess Communication

This and the next chapters deal with middleware:

- This chapter deals with the lower layer of middleware that support basic interprocess communication
- The next one introduces high level communication paradigms (RMI and RPC)



Transmission Control Protocol (TCP) (Connection oriented):

More reliable with more overhead. Data that is sent by producer is queued until receiver is ready to receive them. Sequencing, checking if message delivered. Guarantees message delivery. Flow control.

User Datagram Protocol (UDP) (Connection-less):

Best effort, packet may or may not make it to the other side. No sequencing.

Communication can be synchronous or asynchronous. Talking is synchronous because you send message and wait for response. Asynchronous you send request without waiting for response.

UDP possible failures:

Data corruption: Maybe use checksums to check for this.

Omission failures: buffer full, corruption

Order: messages delivered out of order.

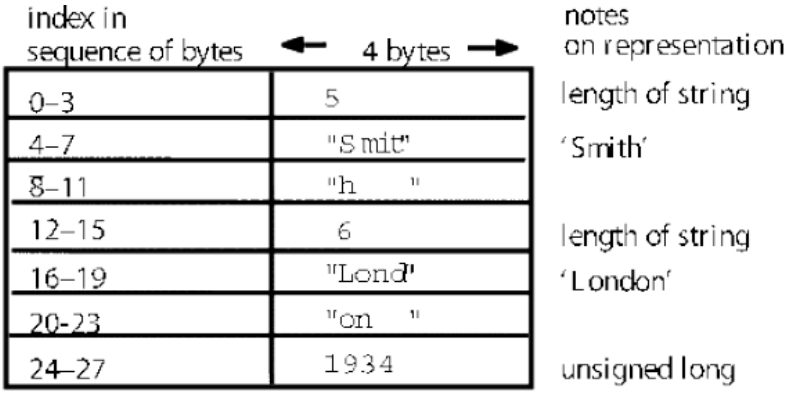
Message Destinations: Messages are sent to an (Internet address, local port) pair.

Socket: End-point for communication between processes. Socket must be bound to a local port to receive messages. It must be listening for connections.

External Data Representations and Marshalling:

- Data structures in programs are flattened to a sequence of bytes before transmission.
- Different computers have different data representations. Need to agree on a standard representation. CORBA, JSON, XML.
- Marshalling: Converting data to form suitable for transmission.
- Unmarshalling: Disassembling data at the receiver.

CORBA's Common Data Representation:



index in sequence of bytes	← 4 bytes →	notes on representation
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h "	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on "	
24–27	1934	unsigned long

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

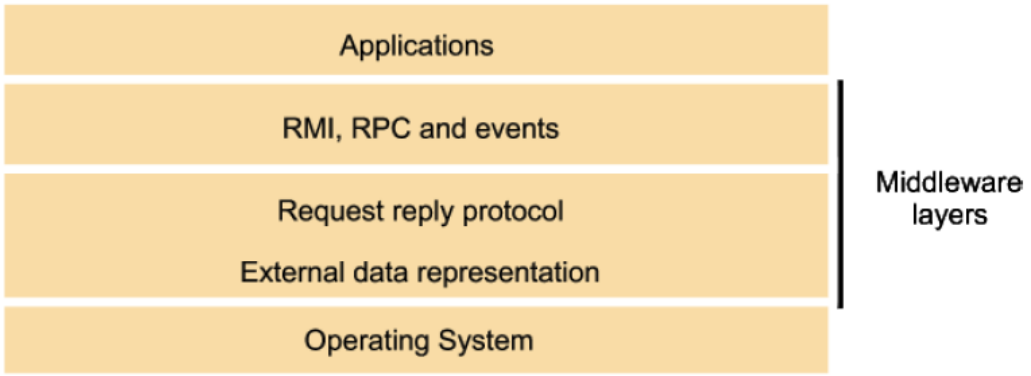
Group Communication:

- A multicast operation allows group communication: Sending a single message to a number of processes identified as group.
- Better performance through replicated data.
- Sender is not aware of the individual recipients.
- Failure model: Omission failures possible.

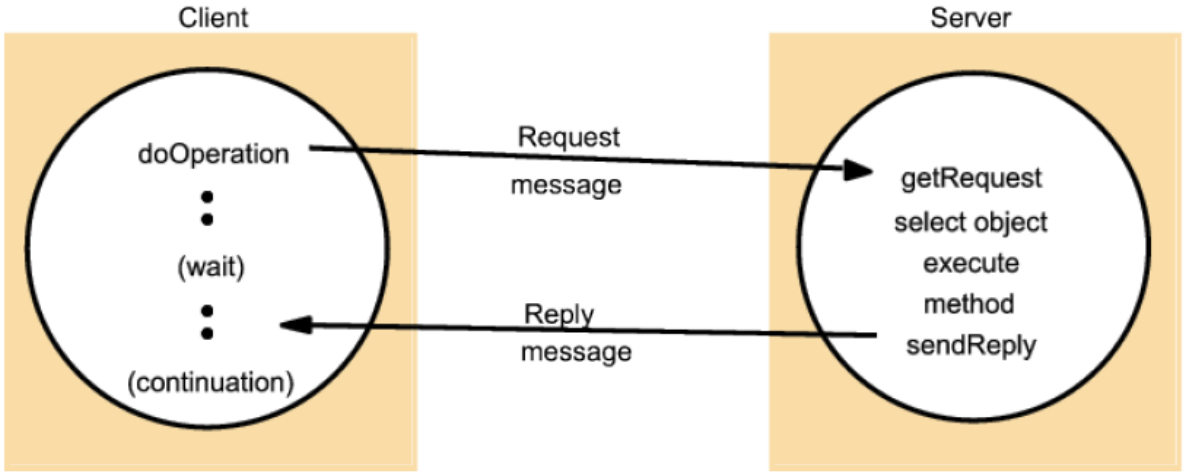
Section 4: Remove Invocation

This section covers the high level programming models for distributed systems. Three widely used models are:

- *Remote Procedure Call model* - an extension of the conventional procedure call model.
- *Remote Method Invocation model* - an extension of the object-oriented programming model.



The Request-Reply Protocol:



Most common exchange protocol for implantation of remote invocation in a distributed system. Do operation, send request, send reply. E.g **client-server**, http.

Design Issues:

- Handling timeouts
- Discarding duplicate messages
- Handling lost reply messages

Exchange protocols

- Three different types of protocols are typically used that address the design issues to a varying degree:
  - the request (R) protocol
  - the request-reply (RR) protocol
  - the request-reply-acknowledge reply (RRA) protocol
- Messages passed in these protocols:

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

E.g for request protocol you just cache a response and if there is none you keep sending the request. Is this good, maybe not.

Invocation Semantics:

Middleware that provides remote invocation has some semantics.

Maybe invocation semantics: the RPC may be executed once or not at all. Unless caller gets results doesn't know if RPC was called.

At-least-once invocation semantics: Either the RPC was executed at least once, and the caller received a response or caller got exception to indicate RPC not executed at all.

At-most-once: RPC either executed exactly once and caller gets a response. Otherwise it was not executed at all and called get exception.

Fault tolerance measures			Invocation semantics
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

CAP Theorem:

When there is a partition there is a tradeoff between consistency and availability.

Aarons Discussion:

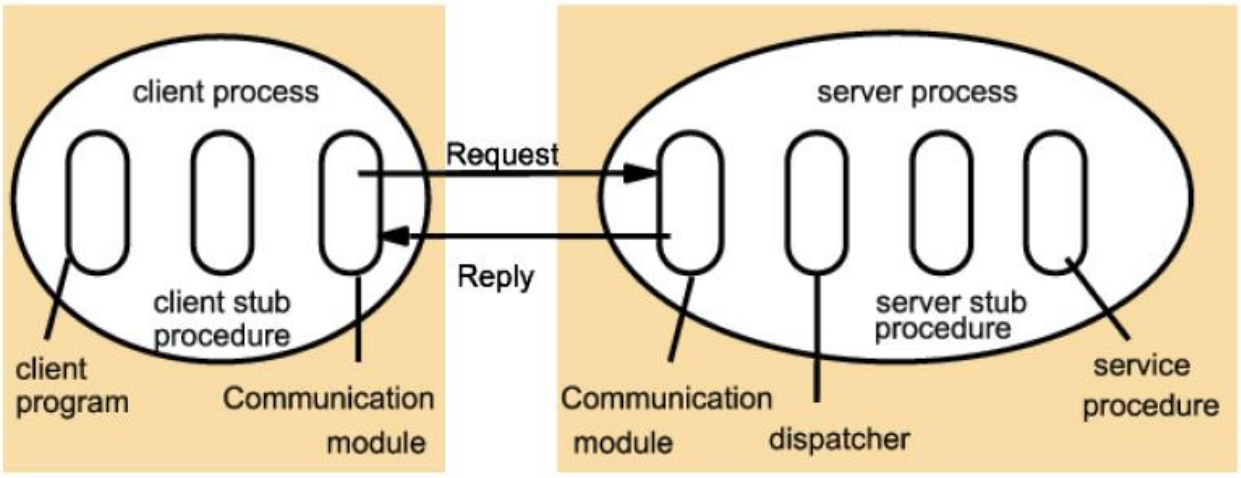
Let's says we have nodes that make up our distributed storage system. Someone makes a read request. The servers then talk to one another to work out where the data for reading can be found. ***The consistency is 100%, if every server manages its own data.***

What if you get a partition of your system, two machines can't talk to each other? Do we want to maintain consistency or availability? If a ***client asks for data if there is network failure*** *how do we get data for this client. You could potentially* ***wait forever if u want consistency****. But then you are* ***not available.***

Maybe ***each server can store caches for any data*** *that they ask from each server. But then you may have a* ***consistency problem*** *because the* ***cached answer may be wrong.***

Eventual consistency send updates of data over time in smaller chunks *and the* ***distributed system can eventually become consistent****. This will* ***minimise failure*** *in the longer term and makes operations of updating not as heavy weight. Can convergence happen faster than data is being updated. It should or its not consistent at all.*

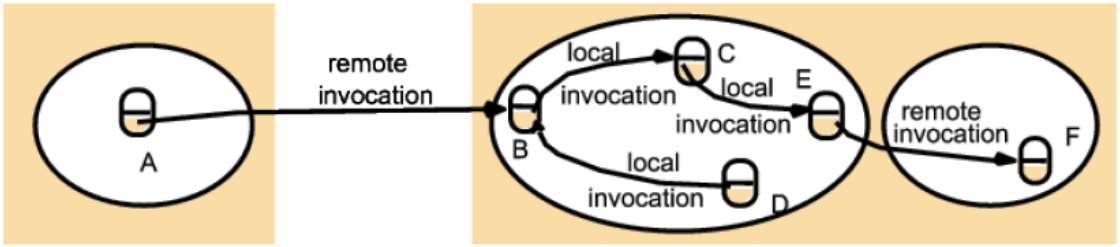
Remote Procedure Call (RPC):



One program calls another procedure on another server. The client thinks it called a local procedure. RPC calls may you have to wait for computation and transition of large objects across network make create bottlenecks. Calls sub-routines instead of objects that have methods.

Remote Method Invocation (RMI) Object Oriented:

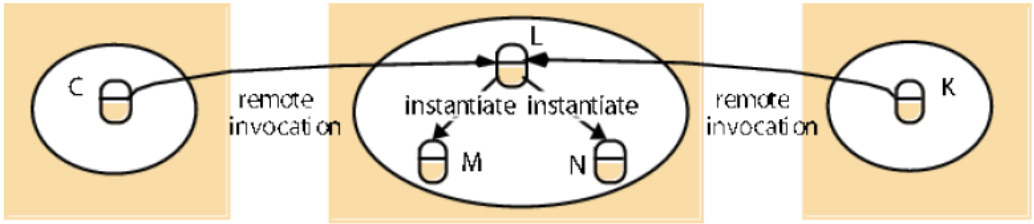
An object that can receive remote invocations is called a remote object. A remote object can receive remote invocations as well as local invocations. Remote objects can invoke methods in local objects as well as other remote objects.



A remote object reference is a unique identifier that can be used throughout the distributed system for identifying an object. This is used for invoking methods in a remote object and can be passed as arguments or returned as results of a remote method invocation.

Better to make one big call and get everything? It's costly to continually use the network for transport of data.

Actions can be performed on remote objects (objects in other processes of computers). An action could be executing a remote method defined in the remote interface or creating a new object in the target process. Actions are invoked using Remote Method Invocation (RMI).

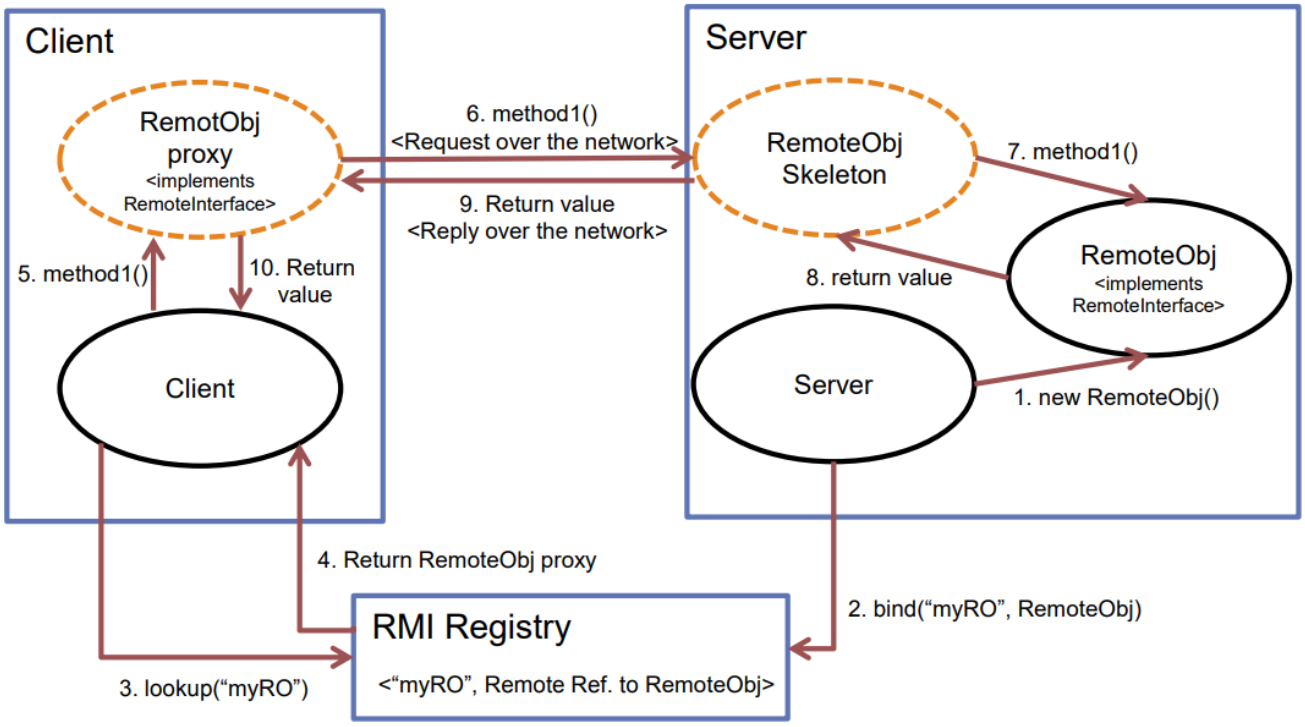


It's not possible in RMI to make a new object in another java virtual machine. You can have another class like a factory that you use to make objects. (Different address space management).

Remote procedure call (RPC) and key components

- **Communication Module**
  Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results
- **Client Stub Procedure**
  Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module
  Unmarshalls the results in the reply
- **Dispatcher**
  Selects the server stub based on the procedure identifier and forwards the request to the server stub
- **Server Stub Procedure**
  Unmarshalls the arguments in the request message and forwards it to the Service Procedure. Marshalls the arguments in the result message and returns it to the client

Java RMI Overview



The RMI software: This is a software layer that lies between the application and the communication and object reference modules. Following are the three main components.

- **Proxy**: Plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
  - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
  - Unmarshalling the results and forwarding them to the invoking object
- **Dispatcher**: There is one dispatcher for each remote object class. Is responsible for mapping to an appropriate method in the skeleton based on the method ID.
- **Skeleton**: Is responsible for:
  - Unmarshalling the arguments in the request and forwarding them to the servant.
  - Marshalling the results from the servant to be returned to the client.

The binder

Client programs require a way to obtain the remote object reference of the remote objects in the server.

A **binder** is a service in a distributed system that supports this functionality.

A binder maintains a table containing mappings from textual names to object references.

Servers register their remote objects (by name) with the binder. Clients look them up by name.

An object that wants to be available for remote invocation needs to register itself to a binder.
JVM's that want to access these objects look up in the registry/binder to find the remote reference. The binder connects jvms to remote objects.

Section 5: Indirect Communication

Indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s).

- *Space uncoupling*: sender does not know or need to know the identity of the receiver(s)
- *Time uncoupling*: sendder and receiver can have independent lifetimes, they do not need to exist at the same time

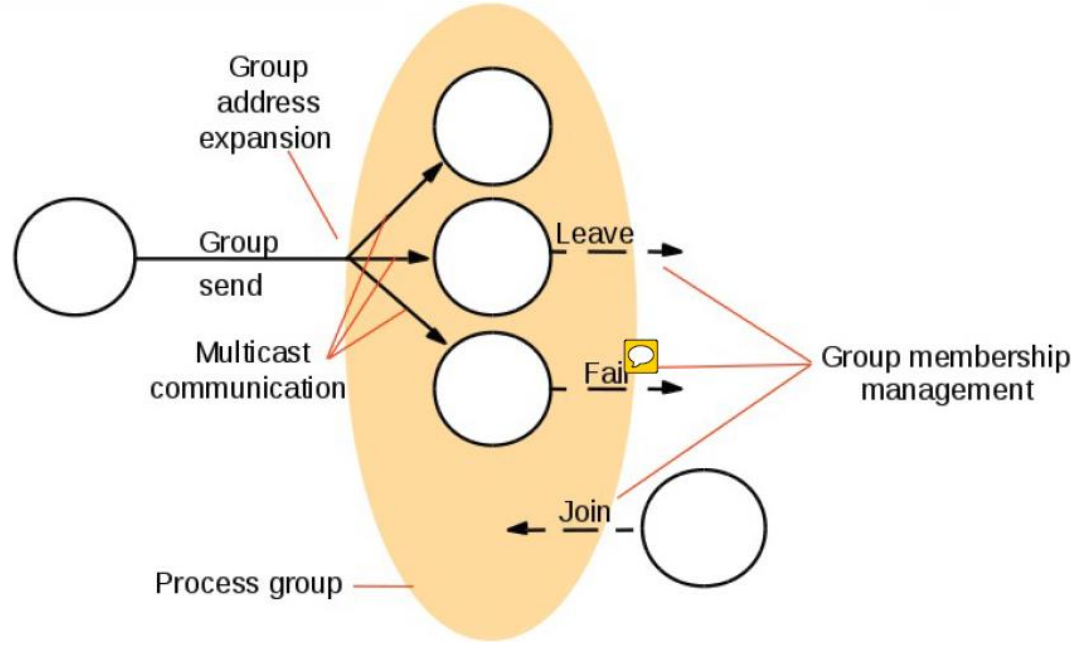
Time uncoupling is not synonomous with asynchronous communication. Asynchronous communication doesn't imply that the receiver has an independent lifetime, in otherwords we could consider a time coupled asynchronous system.

Group Communication:

Offers space uncoupled service whereby a message is sent to a group and then this message is delivered to all members of the group. More than just IP multicast.

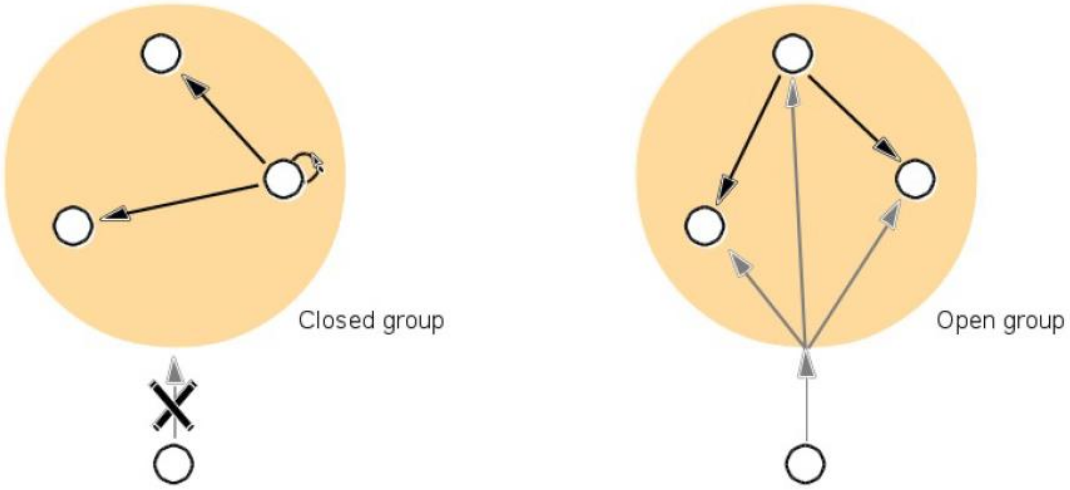
- Manages group membership
- Detects failures and provides reliability/ordering guarantees
- Middlewear has features like join/leave/membership/broadcast

Group Model:



Models help you build you distributed system. If you have a situation where processes leave and join a group, then you can use the group model.

Group services



- closed groups only allow group members to multicast to it
- overlapping groups allows entities to be members of multiple groups
- synchronous and asynchronous variations can be considered

Implementation Issues:

- Reliability and ordering in multicast.
    - Order sender sends needs to be preserved. (casual ordering)
    - Order in which each client sends must be preserved. (total ordering)
- Group membership management
    - Group members leave and join
    - Notify group of member changes

Publish/Subscribe Model:

Event based-system. Publishers publish events to an event service and subscribers subscribe to the events of a particular publisher. Subscribers get notified when new events occur.

Events and Notifications:

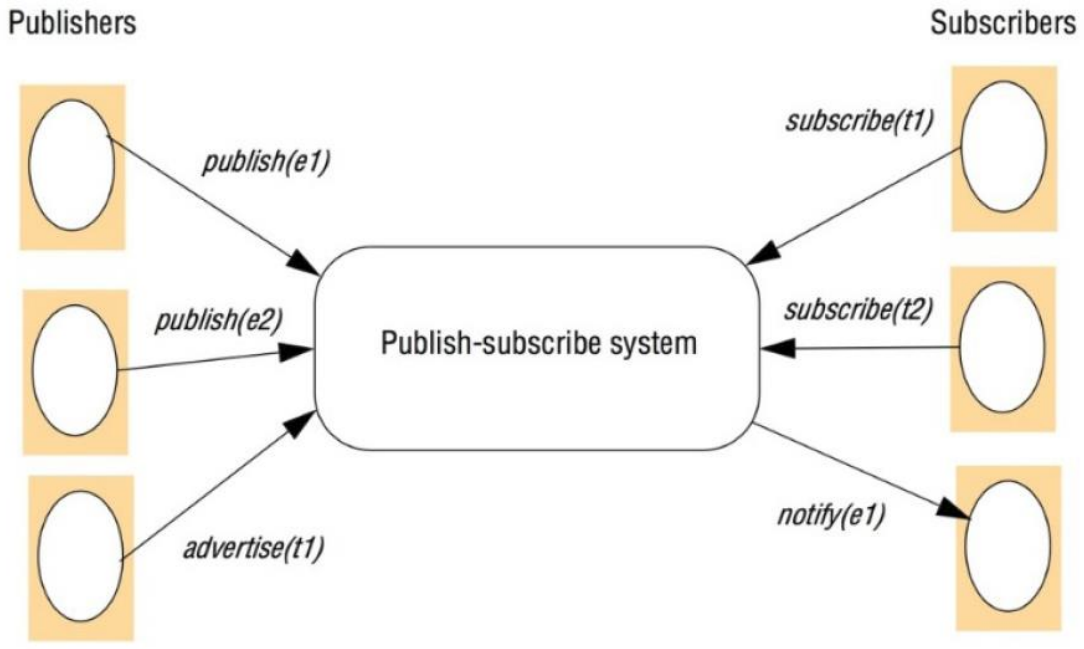
RMI and RPC supports the synchronous communication model where client invoking call waits for the results to be returned.

Events and notifications are associated with the asynchronous communication model. Even based systems can use the publish-subscribe communication model.

Characteristics of event-based systems:

Heterogeneity: Allows objects that were not designed to interoperate to communicate due to loosely coupled nature.

Asynchronous: Communication is asynchronous, and event driven.



Advertise provides an additional mechanism for publishers to declare the nature of future events, i.e. the types of events of interest that may occur.

Types of Publish-Subscribe Systems:

Channel Based: Publishers publish to named channels and subscribers subscribe to all events on a named channel.

Type Based: Subscribers register interest in types of events and notifications occur when particular types of events occur.

Topic Based: Subscribers register interest in particular topics and notifications occur when any information related to the topic arrives.

Content Based: Subscribers specify interest in particular values for multiple attributes. Notifications are based on matching the attribute specification.

Case-study Apache Kafka:

Kafka:
Pub/sub messaging system. Used for real time event processing.
Communication with front end GUI and our back-end.

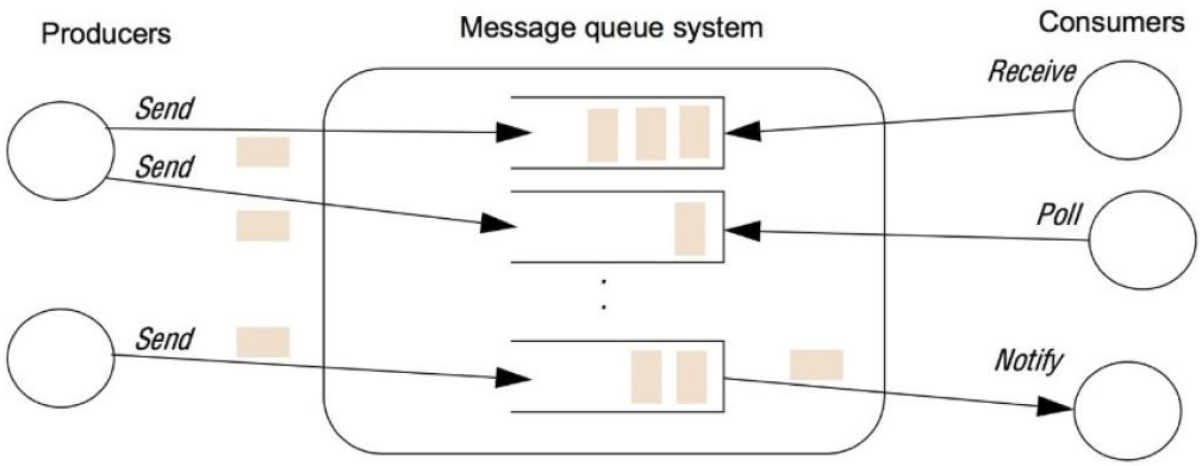
Live services need to present certain information. These live services subscribe to kafka. Kafka updates services/subscribers based on changes. Kafka can talk to hadoop processor to do processing on the data in kafka.

Addressed cap theorem.
They assumed that networking partition is rare. Brokers will run on a data center. SO, they focused on consistency and availability.

Keeping replicas of data across brokers -> availability. Maintain strong consistency between replicas. The leader broker propagates consistency among other brokers.

Exam Q: what is the trade-off of having replicas of data across brokers for availability. Will require additional processing overhead to ensure consistency is maintained.

Message Queues Model:

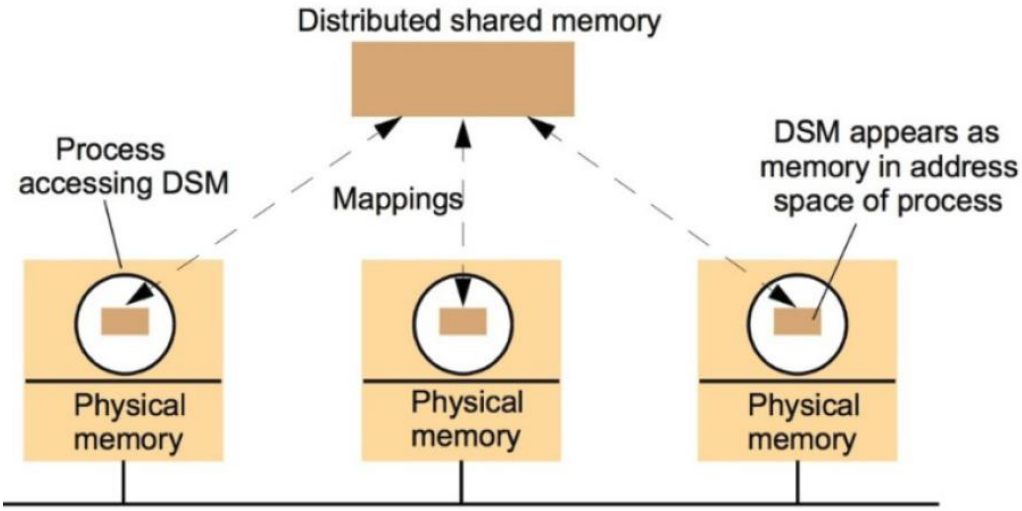


Whereas groups and publish/subscribe systems provide a one to many style of communication, message queues provide a point-to-point service using a message queue as indirection -> space and time uncoupling. Producer adds to queue, consume reads from queue.

Exam Question: Advantage of message queue over publish-subscribe?

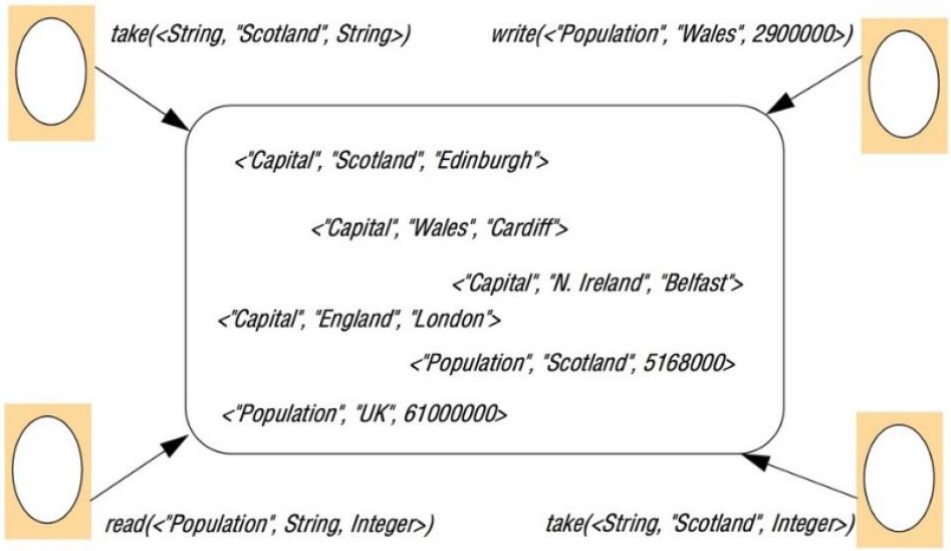
An item in a queue is going only to one user. Many people can read the queue, but each item is unique to a consumer. So, it's better for communicating between particular tasks when there are deliberate things to communicate.

Shared Memory Model:



Sharing data between computers that do not share physical memory. To each process it appears as if the memory is local.

Tuple Space:



Reading and writing to tuple space. Easier to write different version of tuple that deleting tuples?

Overview:

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

Section 6: OS Support

Networking versus Distributed OS:

Networked OS: Support for networking operations. Each host can operate by it self without network environment. Shared files or printer access among multiple computers in a network.

Distributed OS: Abstracts the network from the user so they do not have to specify network commands. One large operating system that manages all distributed resources. May not have everything when disconnected.

Core OS components (all of this is transparent to the user):

Process manager: creation of process.

Thread manager: Handles creation, synchronization, scheduling of threads.

Communication manager: Handles Interprocess communication.

Memory manager: Handles allocation of access to physical and virtual memory.

Supervisor: Handles privileged operations. Those that directly affect shared resources on the host.

Kernel: Part of OS that assumes full access to host's resources. Supervisor mode: can access every resource. User mode: Restricted access.

New Processes in a Distributed System (OS?)

Transfer policy: Determines whether new process is allocated locally or remotely.

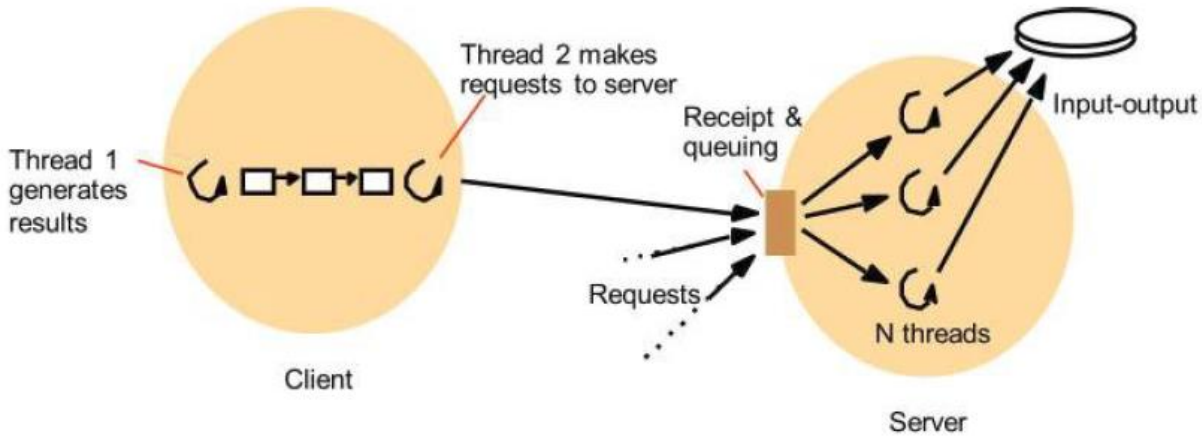
Location policy: Determines which host a process should be allocated on. Adaptive location policies receive feedback about the current state of the DS.

Load manager: Gather information about the current state of the DS. Centralized: Single load manager receives feedback from all other hosts. Hierarchical: Load managers arranged in a tree where internal nodes are load managers and leaf nodes are hosts. Decentralized: Load manager for every host, talk to one another.

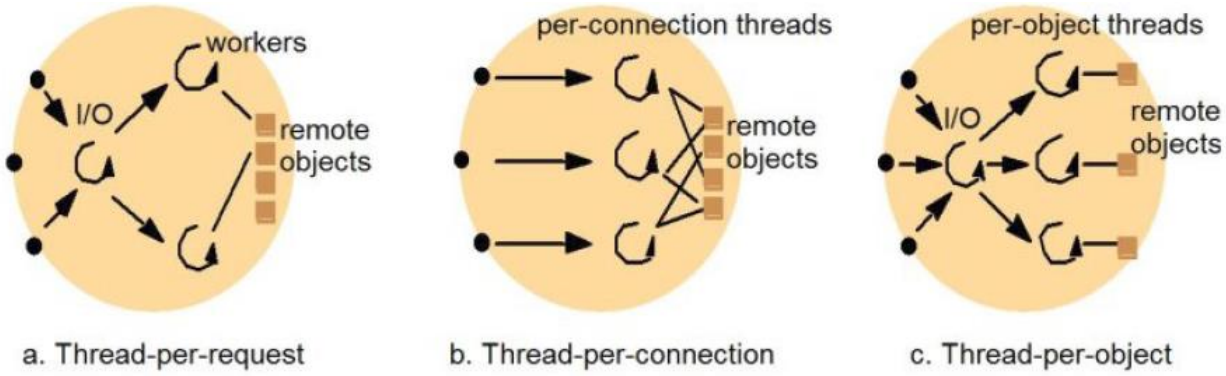
Process Migration:

Processes can be migrated from one host to another by copying their address space. This can be challenging. Process code may be CPU dependent, process may be using host resources such as open files.

Worker Pool Architecture:



- Creating a new thread incurs some overhead that can bottleneck a server. Thus, we can create threads in advance.
- In worker pool architecture, the server creates a fixed number of threads called a worker pool. Requests are queued up and assigned to the next available worker thread. Can use a priority queue.
- If number of workers in a pool is too few, then bottleneck forms at queue.



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Thread-per-request Architecture:

- Separate thread is created for each request and thread is deallocated when finished.
- This allows as many threads as requests to exist. Thread allocation and deallocation incurs overheads. Many threads to manage (overhead).
- Overhead in context(process) switching may outweigh the benefits.

Thread-per-Connection Architecture:

- A single client may make several requests. So, there is one thread for each connection/client. This can reduce overall thread count.

Thread-per-Object Architecture: Worker thread for each remote object/resource

Threads Vs. Multiple Processes: Threads less overhead to allocate/deallocate. Threads are light weight and context switching is cheaper.

Thread Scheduling:

Pre-emptive thread can be suspended at any time. Concurrency issues can be complicated with this feature, but it helps with giving balance in resource usage.

Invocation Performance:

User space procedure: Minimal overhead, stack space.

System call: Overhead is characterized by a domain transition to and from kernel.

Interprocess on same host: There is a domain transition to kernel, to the other process and back to kernel and back to calling process.

Interprocess on a remote host: The same overheads as on same host, plus the overheads of network communication between the two kernels.

Factors that delay RMI:

- Network delay
- Marshalling (converting and copying data)
- Thread scheduling and context switching
- Waiting for acknowledgements.

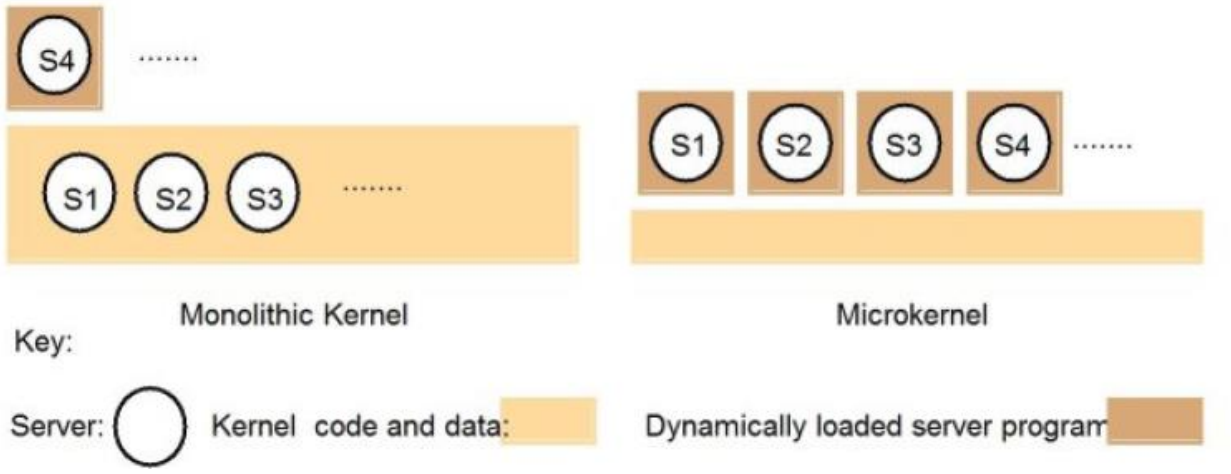
Communication via Shared Memory:

Shared memory can be used to communicate between user processes and between a user process and the kernel. Write to and read from the shared memory. Data is then not copied to and from kernel space. Need synchronization.

Choice of Protocol:

TCP is used when client and server are to exchange information in a single session for a reasonable length of time. Can suffer if endpoints changing identity. UDP is used for request-reply applications that' don't need a session for any length of time.

Monolithic and Micro Kernel Design:



Monolithic Kernel (UNIX OS):

- Procedure calls very fast.
- Diverse functionality provided by kernel. Paging, file system etc. Harder to change kernel functionality as opposed to server functionality.
- If there is a bug in one of the processes, the whole kernel can die.

Micro-Kernel:

- Kernel as small as possible, functionality of the OS provided by servers. Kernel will likely be free of bugs as its smaller.
- Distributed design. Communication with Interprocess communication. Can't make procedure calls.
- May be more efficient since complex functionality is provided through a smaller number of system calls.
- This is good processes can kill themselves but not the whole kernel.

Emulation and Virtualization:

- Adoption of micro kernels is hindered because they do not run software that a vast majority of computer users want to use.
- Virtualization can be used to run multiple instance of virtual machines on a single real machine. Virtual machines can run different kernels.
- An emulator is something that doesn't physically exist but has functionality as if it does. E.g android emulator.

Xen Architecture (Read if CBF):

Hypervisor:

A hypervisor or virtual machine monitor (VMM) is computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating systems may share the virtualized hardware resources: for example, Linux, Windows, and macOS instances can all run on a single physical x86 machine. This contrasts with operating-system-level virtualization, where all instances (usually called containers) must share a single kernel, though the guest operating systems can differ in user space, such as different Linux distributions with the same kernel...

Section 7: Security

Threats and Attacks:

Leakage: Acquisition of information by unauthorized recipients.

Tampering: Unauthorized alteration of information.

Vandalism: Interference with the proper operation of a system without gain to the attacker.

Misuse of communication Channel:

Eavesdropping: obtaining copies of messages without authority.
Masquerading: sending or receiving messages using the identity of another principal without their authority.
Message tampering: intercepting messages and altering their contents before passing them on.
Replaying: storing intercepted messages and sending them later.
Denial of service flooding a channel or other resource with messages to deny access for others.

Threats from mobile code: If you run software from the internet you have to ensure that the software is trusted.

Information Leakage: Even if messages are secure themselves. The meta-data of messages i.e the frequency or the time at which messages are sent can be used for de-anonymization.

Worst-case Assumptions you should have when Designing:

- Interfaces exposed. Socket open to public.
- Networks insecure. Messages can be looked at, copied.
- Attackers have unlimited resources.

Cryptography:

kA -- Alice's secret key.

kB -- Bob's secret key.

kAB -- Secret key shared between Alice and Bob.

kApriv -- Alice's private key (known only to Alice).

kApub -- Alice's public key (published by Alice for everyone to read).

{M}_k -- Message **M** encrypted with key **k**.

[M]_k -- Message **M** signed with key **k**.

Eve -- Eavesdropper.

Mallory -- Malicious attacker.

Sara -- A server.

Given an encryption algorithm, **E**, a decryption algorithm, **D**, a key, **k**, and a message, **M**, **{M}_k=E(M,k)** and **M=D({M}_k,k)**.

If **k=kA** is Alice's secret key then **{M}_k** can only be decrypted by Alice using **k**.

If **k=kAB** is a secret key shared between Alice and Bob then **{M}_k** can only be decrypted by Alice or Bob using **k**.

If **k=kApriv** is Alice's private key, from a public/private key pair, then **{M}_k** can be decrypted by anyone who has **kApub**.

If **k=kApub** is Alice's public key, from a public/private key pair, then **{M}_k** can only be decrypted by Alice using **kApriv**.

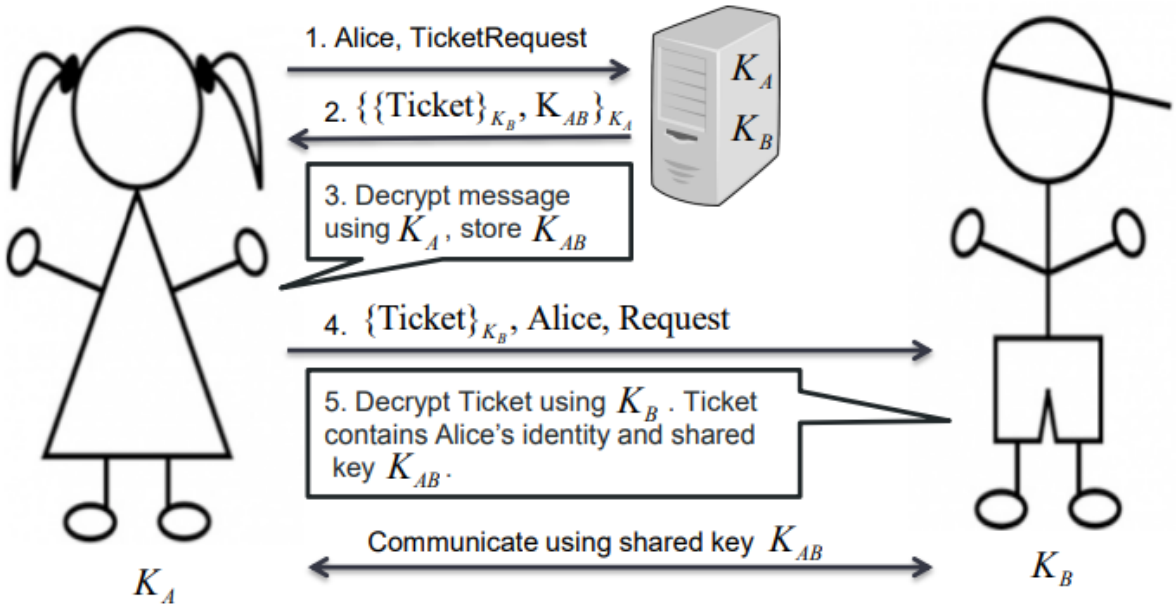
Public/private key encryption algorithms typically require 100 to 1000 times more processing power than secret-key algorithms.

Major Features Encryption Helps Ensure:
Secrecy: Messages encrypted with a key can only be decrypted by recipient who knows the decryption key.
Integrity: Ensure the message hasn't been altered. Use checksums.
Authentication: Message is from the known and trusted sender. Digital signature

Secret Key Cryptography (Symmetric):

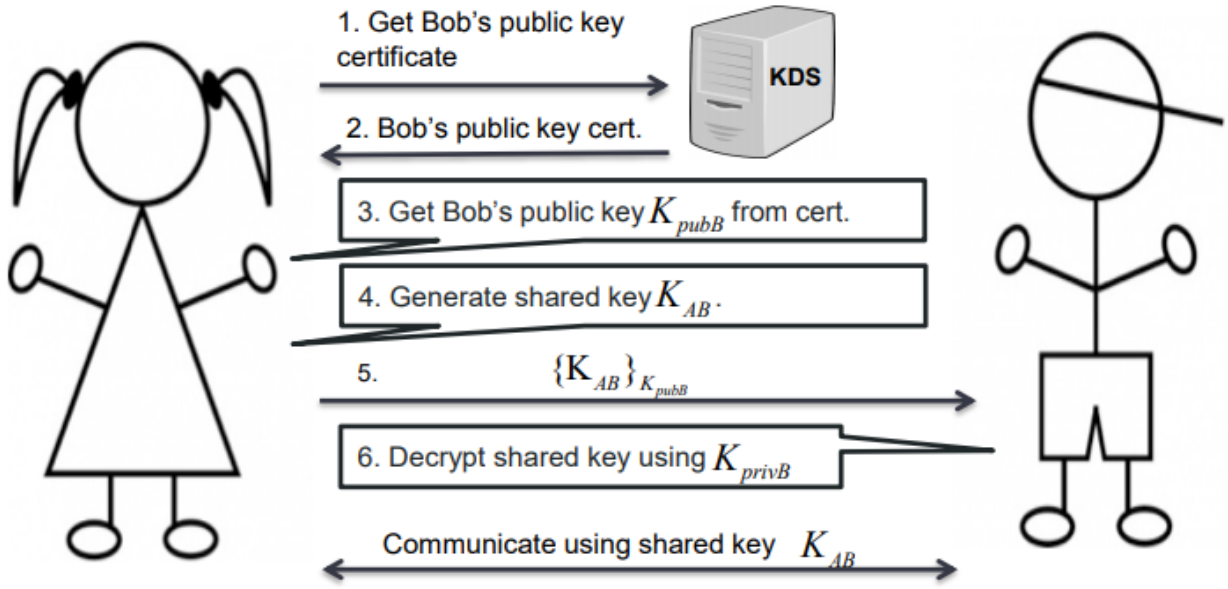
- Alice and Bob can agree on a shared key that is used for encrypting and decrypting.
- If the message makes sense when decrypted or contains some agreed checksum. Bob knows it hasn't been tampered with.
- Some problems:
    - How can Alice and Bob share the key securely.
    - How does Bob know that Mallory didn't make a copy of the message and resent it to Bob? (replaying)

Authentication using an Authentication Server (Symmetric key share):
How can Alice and Bob communicate, such that messages are not tampered with and messages are authenticated.



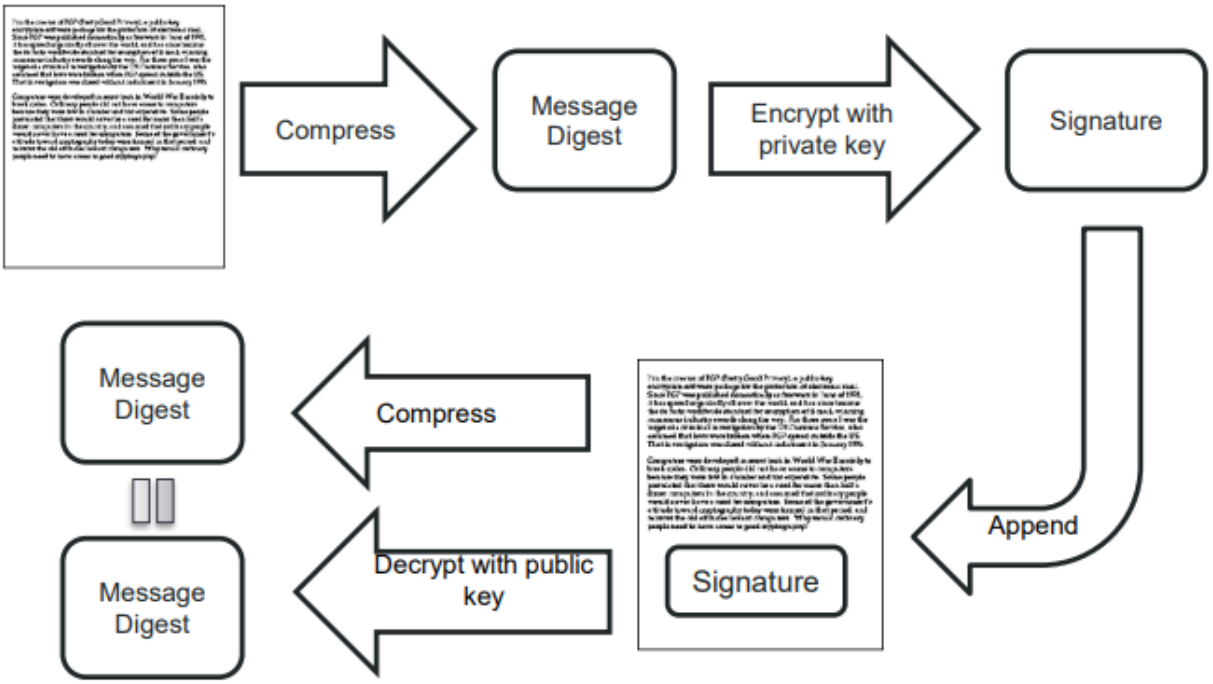
1. Alice sends an unencrypted message to server stating her identity and requesting a ticket for access to Bob.
2. Server sends response to Alice encrypted with her key. The message has a ticket that is encrypted with bobs key and a new secret key **KAB** that can be used for communicating with bob.
3. Alice decrypts the response using **KA**, she can't tamper with the ticket.
4. Alice sends a request to bob. Ticket has Alice's identity and shared key **KAB.**
5. Bob receives ticket which he decrypts using his own key.
6. Now Alice and Bob can communicate with shared key **KAB.**

Authenticated Communication with Public Keys (Symmetric key share):
How can Alice and Bob establish a shared key to communicate secretly using a Key Distribution Service.



1. Alice accesses a key distribution server to obtain a public key certificate giving Bob's public key. The certificate was signed by the server. The server key is widely known and used to check signatures.
2. Alice creates a key and encrypts it using bob's public key and sends it to Bob.
3. Bob decrypts the message and now has **kAB** to use for communicating with Alice.

Digital Signatures:



- Verify to a third party that a message or a document is an unaltered copy of one produced by signer. Assures that any changes made to the data that has been signed cannot go undetected. Difficult to forge signature.

1. Message compressed to a digest.
2. Alice encrypts the digest with her private key, this is the signature.
3. Alice appends this signature to the original message.
4. When bob receives this message + signature he can compress the message and decrypt the signature with Alice's public key.
5. If digest from message = digest from signature, then it's unaltered message made by Alice.

Certificates:

Document containing a statement signed by a certifying authority. Certificates can be used to verify that the public key of someone is in fact theirs. Can use a Merkle tree to make sure all certificates are authenticated.

TLS handshake

Section 8: File Systems

A distributed file system emulates the same functionality as a non-distributed filesystem for client programs running on multiple remote computers. Keep track of files. (A file has data and attributes: type, timestamps, length, access).

Distributed File System Requirements:

Transparency: Access, location, performance and scaling transparency.

Concurrent file Updates: Multiple clients' updates to files should not interfere with each other.

File Replication: Copies of files on multiple servers for availability, backup.

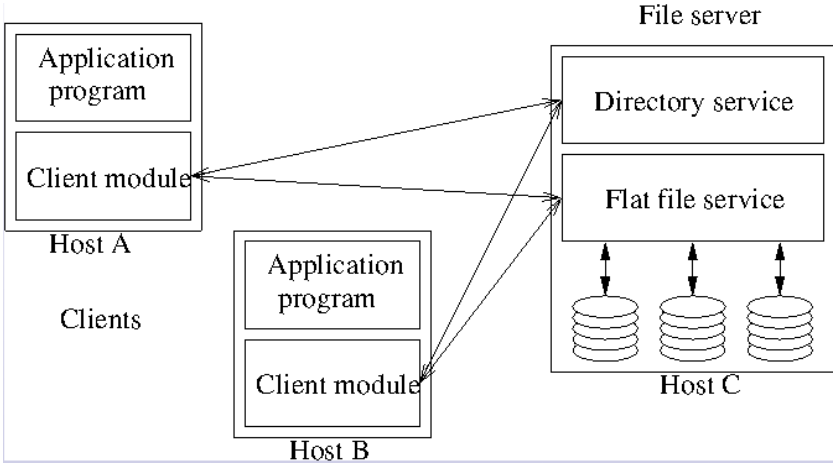
Consistency: Multiple concurrent access to file should see a consistent representation of that file.

Efficiency + Security + Fault Tolerance

File Service Architecture:

Flat File Service: Implements operations on contents of files, uses *unique file identifier* to identify files.

Directory Service: Primary purpose is to perform Mapping between file names and UFIDs. Client of flat file service. Can add remove files from directory.



Client module: Application program that accesses file server. Uses the services.

Flat File Service Interface:

Differs from UNIX interface mainly for reasons of fault tolerance in these ways:

Repeatable operations: Operations are idempotent, allowing the use of at-least once RPC semantics. Can have duplicate requests.

Stateless Server: Flat file service does not need to maintain any state and can be restarted after failure. UNIX file system maintains a file pointer that is used for reading and writing.

Create file method is not idempotent. Other methods are.

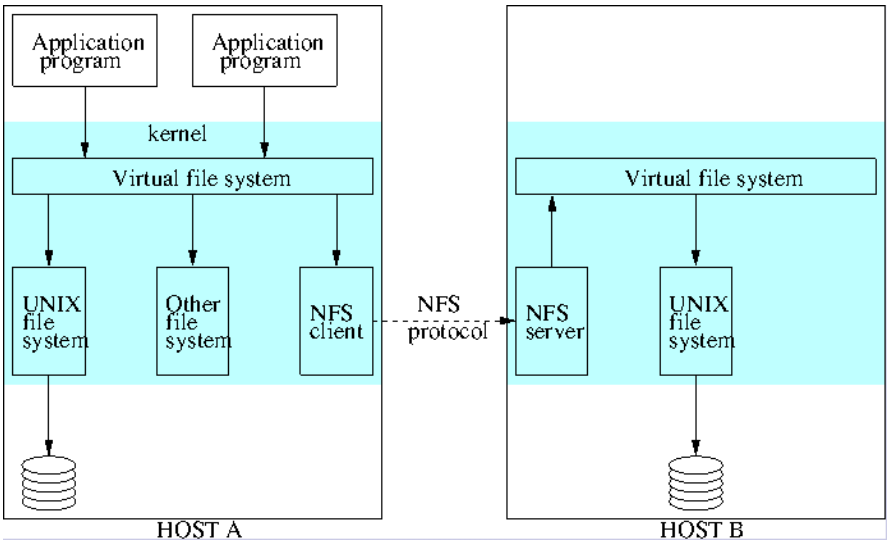
Flat File Service Access control:

The service needs to authenticate RPC callers, ensure all operations are legal. Can give access tokens to authenticated users.

File Group:

Collection of files located on a given server.

Virtual File System:

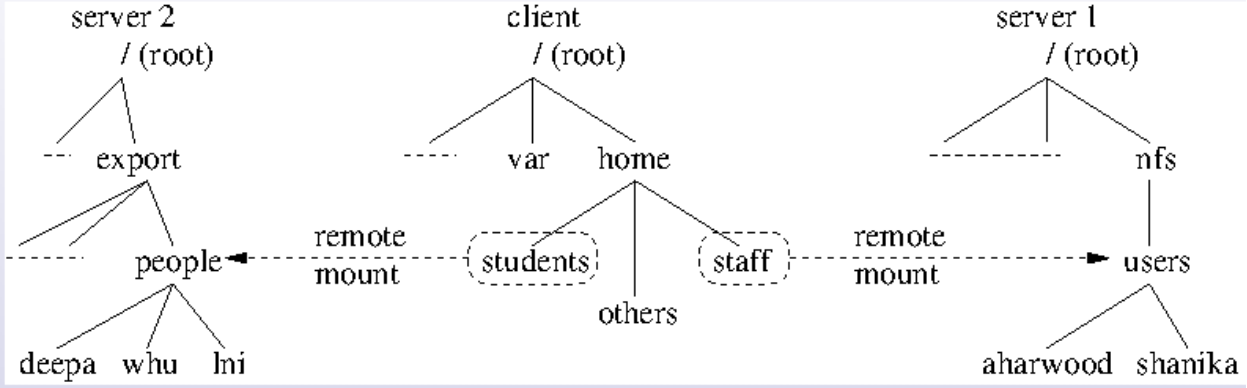


Unix uses a virtual file system to provide transparent access to any number of different file systems. The Network file system operates in the same way.

Client Integration:
NFS client is integrated with the kernel so that: There is access transparency. There is a shared cache. Encryption to handle security.

Server interface:
Integrates both the directory and file operations in a single service.

Mount Service:
Each server maintains a file that describes which parts of the local filesystems that are available for remote mounting.



NFS Mounting:
Have server 1 with local file system. Have client with local file system. Mounting a remote file directory in a local file directory. The staff directory is a mount point for users. When people cd in staff they see these two people here. NFS brings *information transparently*. Distributed file system makes everything look like a local file system by bringing files systems across all the servers to make a big local file system.

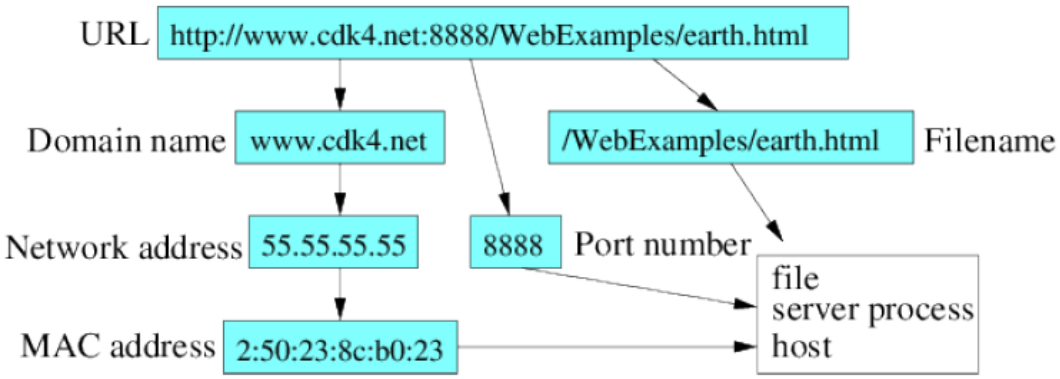
Server Caching:
when u read data from the disk drive you store it in memory. When you write data to the disk drive it. Might not get written to the drive, may just get written to cache. Speeds up reading and writing so you don't have too many disk accesses. Caching would be in the server and in the client. **Read-ahead**, **delayed-write**.

NFS summary

- *Access transparency*: Yes. Applications programs are usually not aware that files are remote and no changes are need to applications in order to access remote files.
- *Location transparency*: Not enforced. NFS does not enforce a global namespace since client filesystems may mount shared filesystems at different points. Thus an application that works on one client may not work on another.
- *Mobility transparency*: No. If the server changes then each client must be updated.
- *Scalability*: Good, could be better. The system can grow to accommodate more file servers as needed. Bottlenecks are seen when many processes access a single file.
- *File replication*: Not supported for updates. Additional services can be used to facilitate this.
- *Hardware and operating system heterogeneity*: Good. NFS is implemented on almost every known operating system and hardware platform.
- *Fault tolerance*: Acceptable. NFS is stateless and idempotent. Options exist for how to handle failures.

- *Consistency*: Tunable. NFS is not recommended for close synchronization between processes.
- *Security*: Kerberos is integrated with NFS. Secure RPC is also an option being developed.
- *Efficiency*: Acceptable. Many options exist for tuning NFS.

Section 9: Name Services



Services may need to co-operate to have name consistency.

Pure name: Must be looked up to obtain address, before named resource can be accessed. Non-pure has information about the object.

URI (uniform resource identifier): Identify resources on the web.

URN (uniform resource names): URI's that are used as pure resource names rather than locators. Requires resolution service or name service to translate the urn to an actual address to find the resource.
Example:
doi:10.1007/s10707-005-4887-8 where the lookup service is http://dx.doi.org/10.1007/s10707-005-4887-8 and this resolves to http://www.springerlink.com/content/c250mn1u2m7n5586/ and in turn this refers to a document, "Building and Querying a P2P Virtual World".

Name Services:
The major operation of a name service is to resolve a name.
Unification: Resources should use the same naming services.
Integration: Sharing resources that were created in different domains requires naming those resources.

Goals of the Global Name Service:
- Handle arbitrary number of names and serve any number of organizations.
- High availability.
- Fault isolation. Local failures do not affect entire service.

Name Spaces: Defines set of names valid for a given service. Can have multiple domain names for one IP address.

Distribution and Navigation:

- Store naming information on a number of different servers.
- Use replication to increase availability.
- Bottlenecks are caused by network I/O.

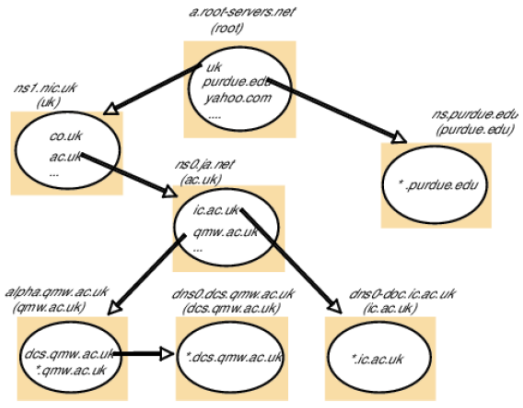
May need to propagate between servers to resolve a particular name. So, we use navigation.

- **Iterative navigation:** Make request to different servers one at a time. Start at root of domain name and resolve to next server in tree.
- **Multicast navigation:** Client multicasts request to the group of servers, only server that holds the name will return a response/result.
- **Non-recursive server-controlled navigation:** Client sends request to server and the server continues on behalf of the client to do iterative navigation.
- **Recursive server-controlled navigation:** Client sends request to server and the server sends request to another server recursively.

Caching: Results of name resolution can be cached by the client and by servers. Can help in server failure situations. But data needs to be updated regularly.

Domain Name System:
Maps domain names to IP addresses. Can find list of servers for a particular domain.

DNS Name Servers:
The DNS database is distributed across a logical network of servers. Handles queries regarding the location of a domain name's various services.



What the advantages of using absolute names as a naming strategy?

An absolute path refers to the complete details needed to locate a file or folder.

Advantages:

- Easy to locate a file with name
- Greater scalability (can store more names)
- Easy to add and delete names

Disadvantages:

- No location transparency
- File is location dependent, can't be moved

What are the advantages and disadvantages of a naming strategy based on mounting points?

Machine creates set of local names which are used to refer to remote locations or mount points. OS must be able to map servers and paths at each mount point.

Advantages:

- Names do not contain information about the file location
- Remote location can change between reboots

Disadvantages:

- Hard to maintain
- Can lead to confusion if two different local names map to same file or system

What are the advantages and disadvantages of using a global name space strategy?

All nodes have identical name space. Path and name of a file on one machine will be the same on every other machine regardless of where file is stored.

Advantages:

- Location transparency
- Naming consistency
- Storage servers can move file around because client will always contact servers to find a file's location

Disadvantages:

- Files are cached by clients (challenge with consistency)
- Can lead to performance problems (if system scales)